AFRL-RI-RS-TR-2016-119

AUTOMATIC VETTING FOR MALICE IN ANDROID PLATFORMS

IOWA STATE UNIVERSITY

MAY 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-119 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
MARK K. WILLIAMS
Work Unit Manager

/ S /
WARREN H. DEBANY JR.,
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2016	FINAL TECHNICAL REPORT	DEC 2013 - DEC 2015

4. TITLE AND SUBTITLE

Automatic Vetting for Malice in Android Platforms

5a. CONTRACT NUMBER
FA8750-14-2-0053

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)
Morris Chang (Iowa State University)
Witawas Srisa-An (University of Nebraska at Lincoln)

5d. PROJECT NUMBER
APAC

5e. TASK NUMBER
IO

5f. WORK UNIT NUMBER
WA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Prime:
Iowa State University
Dept. of Electrical & Computer Engineering
2215 Coover Hall, Ames, IA 50011

Sub:
University of Nebraska at Lincoln
105 Schorr Center
Lincoln, Nebraska 68588-0150

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2016-119

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In Mobile App Security Testing, Dynamic Analysis is needed to deal with certain malware features that reveals itself only at runtime. Dynamic analysis requires a set of inputs to trigger potential malicious targets during the runtime, so that malware can be exposed. In order to generate the inputs (as a sequence of events for Android Apps), we have developed two software tools: JITANA and APEX which are presented and included in this final project report.

15. SUBJECT TERMS

Program Analysis, Inputs Generation, Concolic Execution, Dynamic Analysis, Android Platform

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**MARK K. WILLIAMS**
U	U	U	UU	42	19b. TELEPHONE NUMBER **(315) 330-4560**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

In this work, we have created an on-the-fly hybrid analysis framework that is easy to deploy and can perform large-scale automatic software analysis in real-world enterprises. This is because our analysis process requires no additional software systems. It harnesses the power of generic programming and high-performance graph processing to efficiently perform analysis based on control-flow, data-flow, and point-to information.

We then explored several deploying scenarios including leveraging runtime information to guide concolic execution by APEX, our newly developed concolic execution engine for Android. The main motivation for this activity is to validate a hypothesis that *malicious code exists in rarely executed paths and exploiting JIT compiler information gives us the necessary information to generate sequences to target these paths.* Our evaluation results revealed that despite our best efforts to provide substantial code coverage, about half of malware locations in the engagement apps were not exercised by random and unit testing. We then applied our concolic execution engines to generate inputs to these unexecuted locations. However, our concolic execution engine still faces many challenges including path explosions when dealing with system APIs and library calls. Our current focus is to overcome this issue through automatic modeling of these APIs.

In addition, we applied our framework to identify potential communication channels that can be used by colluding applications to carry out attacks. To demonstrate the scalability of our framework, we performed large-scale analysis of real-world apps including *Facebook* and *Spotify*. We also applied our framework to simultaneously analyze 90 apps in a device for inter-app connections; the analysis only took 10 minutes. This feature can be particularly useful for analysts working in an BYOD environment. Our work also tackled issues raised by the use of reflection by dynamically identifying reflection targets and capturing them for further analysis.

For future work, we hope to continue to develop JITANA and APEX. By having the ability to analyze an entire device instead of a single app at a time, our framework can serve as a foundation for our research groups and other researchers to develop novel techniques for analyzing collections of applications. It also provides a direct pathway to deal with attacks due to inter-app communications. As of now, JITANA is the only program analysis framework capable of such large scale analysis, and APEX is the only concolic execution engine that applies concolic execution to generate event sequences (i.e., most existing concolic execution engines for Android only apply concolic execution starting at event handlers). We will release these frameworks under GPL licenses this summer. We would also like to continue to work with DARPA to further develop our work and we hope that the frameworks that we have created fit into the long term research plan of DARPA.

Also note that we have submitted our work on JITANA to *International Symposium on Software Testing and Analysis (ISSTA)*. The paper as well as the artifact are being reviewed and we should know the decision on April 19th. We will submit our work on reflection to *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* on March 23rd. Our work on APEX will be submitted to *International Conference on Automated Software Engineering (ASE)* on April 29th.

2.0 INTRODUCTION

Software ecosystems involve interacting sets of actors (software components) sitting on top of a common technological platform that together provide various software solutions or services [21]. Smart-mobile application platforms such as Android are examples of such ecosystems. The Android platform provides software engineers with IDEs that help them develop GUIs and skeleton code, and a powerful application framework that they can use to quickly create Java applications (apps) that run on Android devices. In this way, Google, as the Android provider, can collaborate with external developers to build apps with different functionalities, which in turn increases the value of Android. As such, Android is currently the most widely adopted smart-mobile platform in the world.

Unfortunately, Android is also a frequent target of malware authors. As of January, 2014, there were roughly 10 million known malicious Android apps [14] and this number continues to increase. During just the first three months of 2015 there were nearly 5,000 new malicious apps created each day for Android [7]. As the number of known malware increases so does their complexity and sophistication, rendering them very difficult to detect.

To ensure the dependability and security of software ecosystems such as Android, engineers and security analysts need to be able to isolate faults and security vulnerabilities in those ecosystems. Ideally, faults and vulnerabilities in an app should be detected prior to its deployment. As such, software engineers and security analysts use various software assurance processes in an attempt to detect and remove these as part of the software development process.

There are many examples, however, of flaws in current software assurance processes. First, many Android apps suffer from faults and vulnerabilities due to interactions between apps and framework components. A recent study notes that 23% of Android apps behave differently after a platform update [10]. This is because many Android apps depend on other software to operate (e.g., they rely on social media apps for information sharing) [10]. In fact, it has been reported that about 50% of Android updates have caused previously working apps to fail or render systems unstable [10, 11, 30]). There have also been increasing occurrences of collusion attacks, in which multiple apps work together to perform malicious acts [6, 19]. Program analysis techniques that focus on single apps are not effective for detecting faults and vulnerabilities that involve interactions among multiple software components or multiple apps. As such, these elude testing and are discovered after system deployment, increasing both costs and the overall attack surface of the software system. Clearly, approaches that allow engineers and analysts to cost-effectively analyze interactions among software components would allow more faults and vulnerabilities to be detected during the software development process.

As a second example, there have been reports of malicious apps that escaped vetting by Google and have been admitted to its Play store [22, 23, 26]. Google introduced *Bouncer*, a black-box dynamic analysis system that tests for malicious behaviors in submitted apps [23, 26]. This particular approach requires test inputs that exercise most, if not all, entry points to a program and its critical paths in order to be effective [23]. Unfortunately, having powerful test suites alone is not sufficient for revealing complex vulnerabilities because many vulnerabilities can be exercised only with specific inputs or input sequences. Therefore, being *unable to observe program execution and generate appropriate event sequences* greatly limits the vulnerability detection effectiveness of *Bouncer*. As recently as January, 2016, malicious apps had been missed by Bouncer and been distributed through the Play store [1]. Clearly, an hybrid analysis approach that can per-

form on-the-fly code coverage measurement, provide real-time feedback, and generate necessary event sequences to further enhance code coverage could increase the effectiveness of *Bouncer* by targeting parts of an app that have not been adequately exercised.

Recent adoption of *Bring Your Own Device* (*BYOD*) approaches has created an even greater need for security analysts to be able to vet devices efficiently and effectively. In effect, each app on a device must be vetted. Existing program analysis tools are not capable of doing this cost-effectively, primarily due to reliance on techniques that analyze single apps in isolation instead of analyzing multiple apps in unison. Such core analysis engines cause the frameworks in which they are included to be non-scalable, inefficient, and ineffective. In this case, an approach that can simultaneously analyze all apps on a device efficiently can help analysts detect malicious apps on the device more effectively.

In this project, we have made the following three contributions that advance the state-of-the-art in both static and dynamic program analysis.

1. We introduce *JIT-Analysis* (JITANA), a new framework to support static and dynamic program analysis techniques aimed to vet Android applications for the presence of software defects, security vulnerabilities and malicious intents. As shown in Section 5.0, the proposed framework is highly scalable and capable of focusing its analysis efforts on more fruitful program execution paths. We implemented our framework as part of Dalvik, the virtual machine used in Android, so that it can exploit runtime information (e.g., dynamic compilation information) readily available inside the virtual machine without the need of additional software systems.

2. We introduce *Android Path Explorer* (APEX), a concolic execution engine for Android that is capable of creating event sequences and input values that can be used by a program to reach specific paths. As has been reported by event-based testing researchers, generating test cases with adequate coverage is a challenging problem [8]. As such, instead of relying on test inputs to explore execution paths, our approach track branches off of commonly executed paths that have not been explored. It then computes event sequences and input values that can cause the execution of those paths and analyze the recorded traces. Our proposed concolic execution engine serves two purposes. First, it allows us to remove paths that are not feasible due to the absence of inputs. Second, it allows our approach to explore more paths than those exercised by the test cases.

3. We illustrate how the propose frameworks can be used to tackle emerging security challenges. We have developed a set of analyses for statically and dynamically detectable behaviors such as Inter-Application communications and code coverage. The code coverage information provides analysts with "suspicious paths", which are rarely executed paths. The results are then used to guide APEX, our concolic execution engine to further explore and compute concrete values to possibly exercise these suspicious paths. We create a visualization engine that provide real-time feedback of an on-going analysis. We show that JITANA is highly scalable by using it to simultaneously analyze all applications in Android devices and how it can cope with runtime dynamics by analyzing dynamically loaded code.

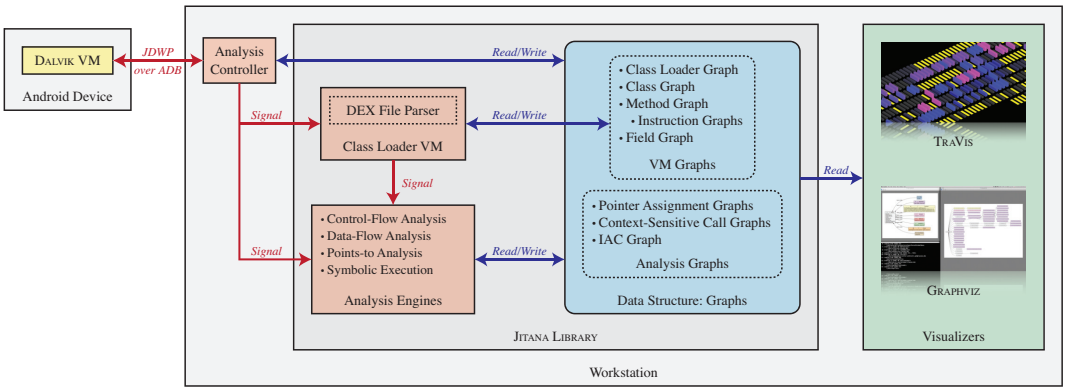


Figure 1: Architecture of JITANA

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we describe the designs and implementations of our two major components: JITANA and APEX.

3.1 The JITANA Framework

Figure 1 provides an architectural view of the JITANA framework. We designed JITANA to be a highly efficient hybrid program analysis framework, so it needs to be able to interface with language virtual machines such as *Dalvik* or the *Android Runtime System* (*ART*). (Our current implementation supports only *Dalvik*.) The interface with *Dalvik* is provided through the *Analysis Controller*, which is connected to *Dalvik* via *Java Debug Wire Protocol* (*JDWP*) over *Android Debug Bridge* (*ADB*). This connection is established primarily for use in dynamic analyses, though it also assists with static analyses in cases in which code is dynamically generated during program initialization (e.g., the Facebook app uses this mechanism during app initialization).

The next major component in the framework is the *Class Loader VM* (*CLVM*). This provides the mechanism used to load classes in an APK along with system related and dynamically generated classes. It is constructed based on the Java Language Specification [20]. Once classes are loaded, the system generates a set of *Boost Graph Library* compliant VM graphs that include *class loader graphs*, *class graphs*, *method graphs*, *instruction graphs*, and *field graphs*. Various *Analysis Engines* then process these to produce control-flow, data-flow and points-to information, which is then fed back in to the VM graphs. Other information is used to construct *Analysis Graphs* such as pointer assignment graphs, context-sensitive call graphs, and an IAC graph. The framework can be integrated with visualization tools such as TRAVIS or run side-by-side with existing visualization tools such as GRAPHVIZ.

Next we describe the design rationales behind and implementation details of major components in the JITANA library.

3.1.1 Design Rationales

To meet our performance and design goals, we implemented JITANA in C++14 instead of an object-oriented language such as Java. We made this choice for several reasons:

- JITANA is designed to work with a virtual machine on a device in various configurations. It can run on a device or workstation as a stand-alone application communicating with a virtual machine in real-time, using an inter-process communication mechanism or a network protocol such as the Java Debug Wire Protocol. It can even be embedded within a virtual machine as a library. Requiring the use of a Java virtual machine in particular would hinder these use cases.

- JITANA analyses may require real-time communication with native applications and the operating system kernel. Achieving this on a virtual machine is difficult, if not impossible, without using the Java Native Interface; however, that would be an expensive bridging approach.

- C++ supports the use of the generic programming paradigm with templates. This paradigm separates algorithms and data structures by defining what is called a *concept*, a description of both syntactic and semantic requirements for one or more types [28]. An algorithm operating on a concept needs to be implemented only once, the same implementation can then be reused for any concrete type that is a model of the concept. The C++ template instantiation mechanism coupled with compiler optimizations has been shown to generate code as efficient as hand-tuned FORTRAN [9].

To elaborate on this last point, the use of *concepts* in generic programming differs from the use of traditional object-oriented techniques in static analysis tools such as SOOT and LLVM. We illustrate the practical difference between generic programming (GP) and object-oriented programming (OOP) through a simple algorithm `max`, which returns the larger of two values.

```
template <typename T> requires TotallyOrdered<T>()
inline T& max(T& x, T& y) {
    reutrn (x < y) ? y : x;
}
```

Listing 1: Generic Algorithm `max` in C++ with Concepts

```
public static Comparable max(Comparable x, Comparable y) {
    reutrn (y.compareTo(x) < 0) ? y : x;
}
```

Listing 2: Algorithm `max` in Java without Generics

In GP, the algorithm can be implemented as shown in Listing 1. This implementation can operate on any type as long as it models the `TotallyOrdered` concept, which requires operators $<$, $>$, $\leq$, and $\geq$ to be defined with the following semantic constraints:

$$a > b \Leftrightarrow b < a, \quad a \leq b \Leftrightarrow \neg(b < a), \quad a \geq b \Leftrightarrow \neg(a < b)\,.$$

In OOP, the same algorithm can be implemented as shown in Listing 2. It is less reusable than the GP version because it requires the type to be inherited from a specific type named `Comparable`.

As shown in Figure 1, JITANA separates data structures and algorithms. It represents programs as well-defined "Graph Data Structures", and all "Analysis Engines" work on these. The core analysis algorithms are reusable and flexible because they are defined on concepts rather than concrete types. The data structures are also defined to be similar to those used in the actual virtual machine to reduce the overhead of exchanging dynamic information.

3.1.2 Graphs

Most of the data structures used in JITANA are represented as graphs. Typically, a node in such graphs represents a virtual machine object (e.g., a class, a method, an instruction) together with analysis information (e.g., execution counts), while an edge represents a relationship between two nodes (e.g., inheritance, control-flow, data-flow).

Every graph used in JITANA models appropriate graph concepts defined in the Boost Graph Library (BGL),[†] a de facto generic C++ graph library. This means that implementations of highly optimized generic graph algorithms are already available for use by applications on the graphs defined in JITANA without modifications. It also means that new algorithms can be implemented for a concept, rather than for a specific type, so that they can be used with any types modeling the same concept. Analyses can also be performed on any machines on which the BGL library is installed.

Table 1. Types of Handles in JITANA

	DEX Handle	JVM Handle
Class Loader	`struct class_loader_hdl { uint8_t idx; };`	
DEX File	`struct dex_file_hdl { class_loader_hdl loader_hdl; uint8_t idx; };`	N/A
Type	`struct dex_type_hdl { dex_file_hdl file_hdl; uint16_t idx; };`	`struct jvm_type_hdl { class_loader_hdl loader_hdl; std::string descriptor; };`
Method	`struct dex_method_hdl { dex_file_hdl file_hdl; uint16_t idx; };`	`struct jvm_method_hdl { jvm_type_hdl type_hdl; std::string unique_name; };`
Field	`struct dex_field_hdl { dex_file_hdl file; uint16_t idx; };`	`struct jvm_field_hdl { jvm_type_hdl type_hdl; std::string unique_name; };`

In contrast, most existing tools do not use explicit graph types for their data structures. For example, SOOT and LLVM follow the traditional object-oriented approach: an object holds pointers to other objects to imply relationships that implicitly form a graph data structure. This means that algorithms implemented in these platforms are strongly tied to a tool's design details, not to a graph concept. As a consequence, even a simple algorithm such as a depth-first search algorithm must be implemented each time it is needed, and when a new tool comes along, a library of algorithms must be rewritten.

A *handle* is used to identify a virtual machine object. Table 1 lists the handle types most frequently used in JITANA. A handle is small in size, but unlike pointers it does not change values over different executions; this allows us to treat handles as statically unique identifiers. The use of handles allows the graphs generated by JITANA to be persisted and reused. These graphs can also be replicated for parallel analysis on computing clusters.

Table 2 lists some of the graph types used in JITANA. There are two categories of graphs: *virtual machine (VM) graphs* and *analysis graphs*. Virtual machine graphs are graphs that closely reflect

[†] `http://www.boost.org/doc/libs/develop/libs/graph/doc/`

Table 2. JITANA Graphs

Name	Type	Node	Edge
Class Loader Graph	VM Graph	Class Loader	Parent Loader
Class Graph	VM Graph	Class	Inheritance
Method Graph	VM Graph	Method	Inheritance, Invocation
Field Graph	VM Graph	Field	
Instruction Graph	VM Graph	Instruction	Control Flow, Data Flow
Pointer Assignment Graph	Analysis Graph	Register, Alloc Site, Field/Array RD/WR	Assignment
Context-Sensitive Call Graph	Analysis Graph	Method with Callsite	Invocation
Inter-Application Communication Graph	Analysis Graph	Class Loader, Resource	Information Flow

the structure of Java virtual machines. A node in a virtual machine graph represents a virtual machine object (e.g., class, method) that can be created or removed only by the CLVM module in JITANA (described in Section 3.1.3). Modification of a node property by an analysis engine is allowed and is one of the primary ways to track dynamic information such as code coverage. The edge type is erased with the `Boost.TypeErasure` library[‡] so that analysis engines can add edges of any type. Examples of these graphs rendered with GRAPHVIZ, an open-source graph visualization tool, are shown in Figure 2.

Figure 2(a) displays a *class loader graph* for a case in which JITANA analyzes four applications simultaneously. Each class loader is assigned a unique ID (integers in the upper left corners) so that classes with a same name from different applications can be distinguished. For example, both *Facebook* and *Instagram* ship a class named `Landroid/support/v4/app/Fragment;`[§] with different method signatures (i.e., different implementations) because the Facebook app is obfuscated with PROGUARD.[¶] *Class Loader 0* is the system class loader and is used to load important system classes. Each directed edge shows the parent/child relationship between two class loaders (e.g., the system class loader spawns off application class loaders).

Figure 2(b) displays a *class graph* that shows relationships between four classes; the directed edges display subclass relationships (e.g., `Lcom/instagram/.../LoadImageTask;` is a subclass of the abstract class `Landroid/os/AsyncTask;`).

Figure 2(c) displays a *method graph* that shows relationships among several methods within a set of analyzed applications. Nodes represent methods, and edges indicate whether method calls are *direct* or *virtual*. The numbers in the upper left corners of the nodes indicate the applications to which the methods belong.

Figure 2(d) illustrates an *instruction graph* for a method. It includes both control-flow (solid edges) and data-flow (dotted edges) information. The data-flow information is derived via reachability analysis performed on virtual registers.

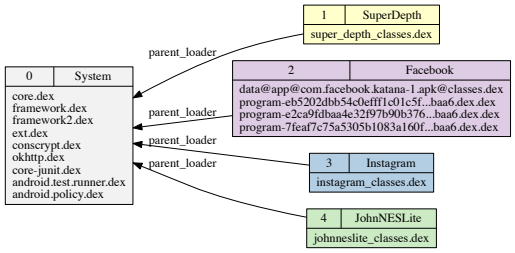
Not depicted in the figure is a *field graph*. JITANA stores a list of fields as nodes, but by default it does not add edges to this graph. This data can still be used for analysis purposes.

Analysis graphs are used by analysis engines to represent relationships that cannot be expressed

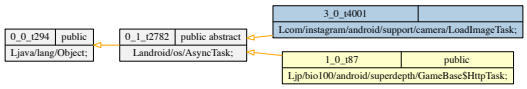
[‡] `http://www.boost.org/doc/libs/develop/doc/html/boost_typeerasure.html`

[§] A class name in a JVM starts with '`L`' and end with a semicolon '`;`'.

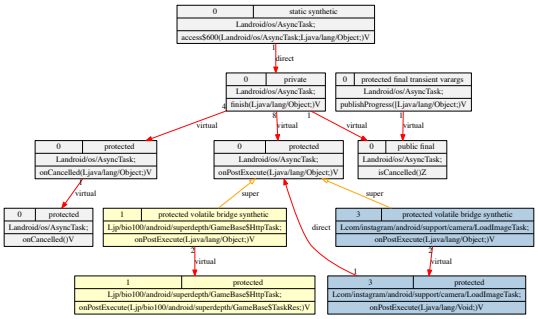
[¶] `https://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920`



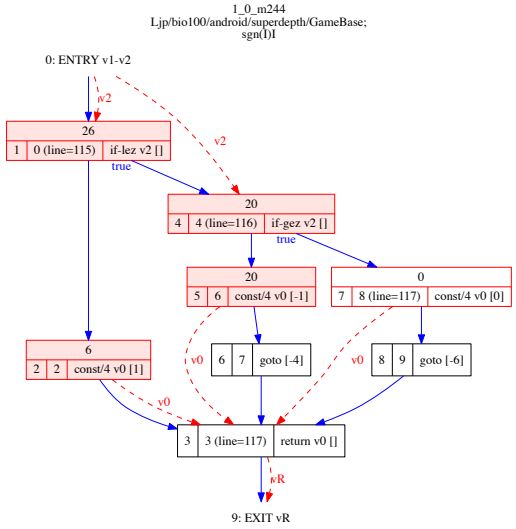
(a) Class Loader Graph



(b) Class Graph (Subgraph)



(c) Method Graph (Subgraph)



(d) Instruction Graph for `GameBase.sgn(int)` in *SuperDepth*

Figure 2: Illustrations of Various VM Graphs

by virtual machine graphs. For example, the points-to analysis engine (described later on) defines a pointer assignment graph with multiple special node types to represent the flow of pointer values. The CLVM module does not read or write to these graphs. As such, JITANA does not enforce any requirements on analysis graphs, but all analysis graphs defined within JITANA model a set of appropriate graph concepts defined by the Boost Graph Library so that existing generic graph algorithm implementations can be used on these graphs.

3.1.3 Class Loader Virtual Machine (CLVM)

Based on the Java Virtual Machine Specifications [20], a class must be loaded by a class loader. A class loader is a Java class inherited from an abstract class `Ljava/lang/ClassLoader;`; it uses a delegation model to search for classes. Each instance of `ClassLoader` has a reference to a parent class loader. When a class loader cannot find a class it needs to load, it delegates the task to its parent class loader. Both the virtual machine and the Java code running on it participate in this process. In the Dalvik virtual machine, this process occurs as shown in Algorithm 1. Note that a class must be loaded from a DEX file on the file system in the Dalvik virtual machine.

Data: N: name of the class to be loaded, L_{init}: initiating class loader, and $\mathbf{D}_L$: an ordered set of DEX files for a class loader L.
Result: $\langle L_{\text{def}}, C \rangle$: a pair of defining class loader and pointer to a class or interface loaded.
begin
  $L \longleftarrow L_{\text{init}}$;
  $C \longleftarrow \mathbf{null}$;
  do
    foreach $D \in \mathbf{D}_L$ **do**
      if $N \in$ *class definitions list of* D **then**
        $C \longleftarrow$ address of loaded class;
        return $\langle L, C \rangle$;
      end
    end
    $L \longleftarrow$ parent loader of L;
  while $L \neq \mathbf{null}$;
  return $\langle L, C \rangle$;
end

Algorithm 1: Class Loading Algorithm

In JITANA, all pointers to classes are replaced by edges, rendering relationships explicit. As a result, the implementation of Algorithm 1 is merely an instantiation of the depth-first visit algorithm provided by the BGL, requiring the implementation only of the actual loading of a class from a DEX file.

3.1.4 Analysis Engines

A primary difference between JITANA and SOOT is that JITANA is built to be efficient at both static and dynamic analysis. As such, common analysis building blocks such as control-flow, data-flow, and points-to graphs can be annotated on the fly based on incoming runtime events. Next, we describe the approaches used to construct JITANA's analysis engines.

Control-Flow Analysis. The intraprocedural control-flow edges in instruction graphs (see Figure 2(d)) are created by the DEX file parser as it creates instruction nodes. Branch and jump targets are simply encoded as offsets from the DEX instruction. The absence of indirect addressing mode for jumps in the DEX instruction set renders intraprocedural control-flow analysis trivial.

Our interprocedural control-flow analysis includes both direct call edges and virtual call edges in a method graph as shown in Figure 2(c). However, the actual target of a virtual call edge cannot be accurately computed without consulting the *virtual dispatch tables* (*vtables*) that are used to support late-binding features such as inheritance. As such, our analysis is not sound because it ignores the vtables. Furthermore, our analysis is incomplete because it ignores reflection. These two issues are common among static program analysis tools for Java.

To improve the soundness and the completeness of its analysis, JITANA applies additional static analyses such as points-to analysis to determine the actual type of a method's receiver. It can also incorporate dynamic execution information from the virtual machine on the device to identify reflection targets and annotate graphs on the fly.

Data-Flow Analysis. JITANA provides a few data-flow analysis engines. One common analysis supported is reaching definitions analysis, used to generate def-use pairs of registers in the instruction graph as shown in Figure 2(d) (dotted edges). The monotone data-flow algorithm used in the reaching definitions algorithm is implemented as a generic function, and therefore, can be used to generate different types of data-flow analyses such as available expressions or live variable analysis simply by defining appropriate functors. It also works on any graph types that model the concepts required for the control-flow graphs.

In addition to static data-flow analyses, JITANA can incorporate information from the virtual machine. For example, a dynamic taint analysis can be performed on the virtual machine to track data flows from sources to sinks. The results of this analysis can be rendered as edges on a static data-flow graph to provide a more meaningful view of the flow of data.

Points-to Analysis. In Java, most function calls are made using a dynamic dispatch mechanism. Therefore, knowing the actual type of an object in a pointer variable is essential for any interprocedural analysis. JITANA provides a points-to analysis engine. The algorithm is similar to the one used by SPARK [16], a points-to analysis framework in SOOT, with the following differences:

- It uses register def-use information from the data-flow analysis engine to add flow sensitivity. This improves the precision of the analysis, especially because the same register may be reused within a method in the Dalvik architecture.

- It operates on a pair of graphs: a pointer assignment graph (PAG) and a context-sensitive call graph (CSCG). The PAG is conceptually the same as the one used in SPARK, except it is defined as a BGL graph in order to use existing generic algorithm implementations. The CSCG is a call graph specific to a given set of entry points. These graphs are provided as input to the analysis along with entry point information. It is common to have multiple entry points executed in sequence in event-based Android applications; in these cases, the points-to analysis may be called multiple times for each entry point on the same pair of graphs.

3.1.5 Virtual Machine Modifications

JITANA and the Dalvik VM are connected using the Java Debug Wire Protocol (JDWP) over the Android Debug Bridge (ADB). The JDWP is a standard protocol for attaching a debugger to a

virtual machine. With its pre-defined commands, we can observe and control program execution.

These pre-defined commands, however, are not sufficient for all analyses. For example, we need to add a new command to retrieve code coverage information. With our modified virtual machine, the number of executions for all basic blocks of the non-system DEX instructions are counted automatically in interpreter mode; the new command dumps the delta of the execution counters.

This particular modification to the virtual machine is minimal: 127 lines of C++ and 66 lines of ARM assembly code were added. The C++ code handles the additional JDWP communication. It also allocates the same amount of virtual memory pages for the counters when a DEX file is mapped to the memory and records the offsets between them. The ARM assembly code added to the interpreter increments the counter whenever a jump or branch instruction is executed. The address of the counter is given by adding the address of the DEX instruction and the offset to the counter pages.

The code generated by the just-in-time compiler for hot traces remains unmodified; this renders the overhead of the modifications unnoticeable to the user. The Dalvik VM executes hot traces in interpreter mode in the entry even if their compiled code is on the code cache, so we can still obtain correct code coverage data and note the relative hotness of a trace.

3.2 The APEx Framework

The APEx framework combines gray-box GUI testing with concolic execution to find valid event sequences for specific target locations in the code. The overview of APEx can be seen in Figure 3. The first step of the input generation process is a depth first GUI traversal that dynamically builds a GUI model and event-handler map. We then analyze previously uncovered program paths, identify important paths, then use concolic execution to generate event sequences to those paths.

3.2.1 GUI Exploration

Our GUI exploration uses a depth first strategy to traverse GUI layouts and exercise relevant events in each layout until all the layouts and events are explored. The work flow of GUI traversal is shown in Figure 4. Events are generated in a gray-box approach. *AndroidManifest.xml* can provide information such as package name, activity class names, *MainActivity* name, and intent filter that can be used to perform automatic GUI traversal. The GUI traversal process is built on the *UIAutomator* program in Android SDK. The *UIAutomator* can take screen shots and dump current layout hierarchy of an Android device at any given time. By checking layout hierarchy and applying events in a lock-step manner, the GUI traversal process keeps exploring new layouts and record the layout transitions until all layouts and all events are explored. The GUI traversal process generates two important data: the GUI model, and the Event-Handler map.

GUI model contains all the layouts and events explored during the traversal. Each layout information is stored as a node, and each event is stored as a directed edge that starts from the layout node before applying the event and ends at the layout node after applying the event. The layout hierarchy information retrieved from *UIAutomator* contains various attributes of all the *View* objects that are showing on the device screen. Such information is used to: (i) create layout nodes, (ii) find applicable events for each *View* object, and (iii) compare layouts before and after applying an event. The GUI model is generated in a depth-first manner. The traversal keeps exploring layouts and events until arriving an layout where all the events keep the same layout. Then the

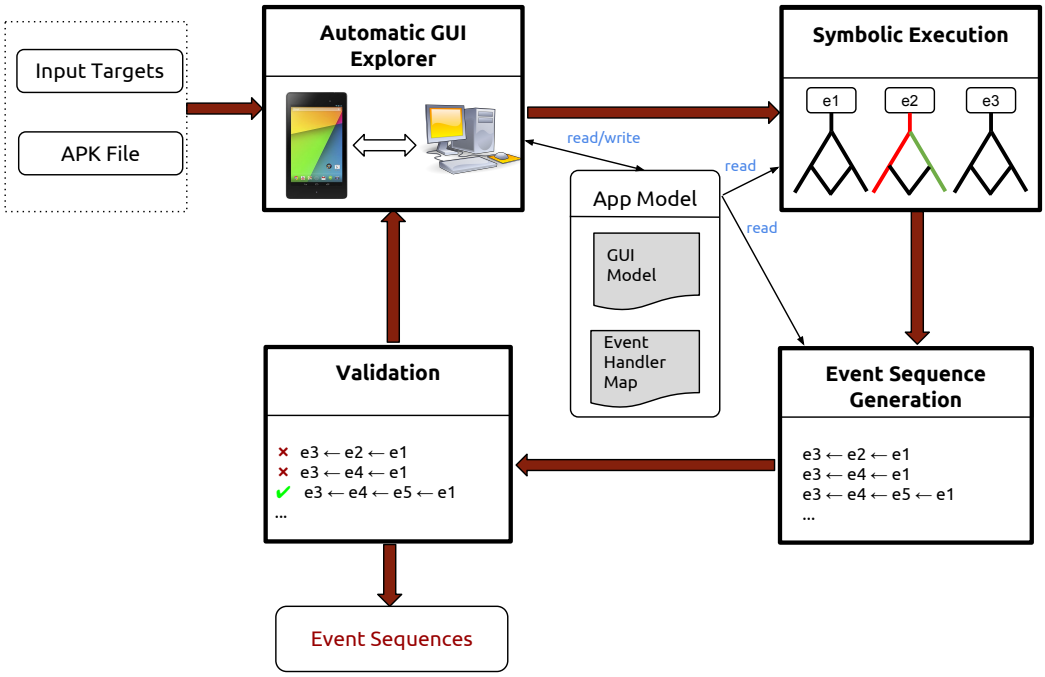


Figure 3: Overview of APEx Framework

app restarts and revisit the most recent layout that still has unexplored events, and continues the traversal process. The GUI traversal finishes when there is no unexplored events in any layout in the GUI model. It is worth noting that the GUI model generated from this traversal stage might be incomplete due to the dynamic nature of our approach. It is possible that certain GUI layouts can only be triggered by certain complex event sequences. Such GUI layout will be missing in our GUI model. We compensate this possible drawback in the later stage by identifying uncovered GUI transition statements and generating event sequences to trigger the GUI transition.

Event-Handler map pairs an event to an event handler method (e.g., *Button1* to *onClick1()*). Instrumentation is used to monitor and capture the mapping during GUI traversal. Any method in the app that has the signature of an event handler is instrumented to print out a message when it starts executing and when it returns. When an event is applied during the GUI traversal, the corresponding event handler method information is printed out to system console. Usually this event handler registration information can be obtained statically from the layout XML files or the Java code. However there exist certain cases where this information cannot be easily obtained statically. For example, *View* objects can be defined and created during runtime rather than being predefined in the XML or Java code, it is possible that rather complicated static analysis is needed to determine their life cycles and which layouts these *View* objects belong to. Therefore in our GUI traversal process, the events and their registered event handler methods are all discovered dynamically.

3.2.2 Symbolic Execution on Dalvik Bytecode

Based on the Event-Handler map provided by GUI traversal, symbolic execution is performed on each event handler method. Symbolic execution tries to traverse all the execution paths caused by

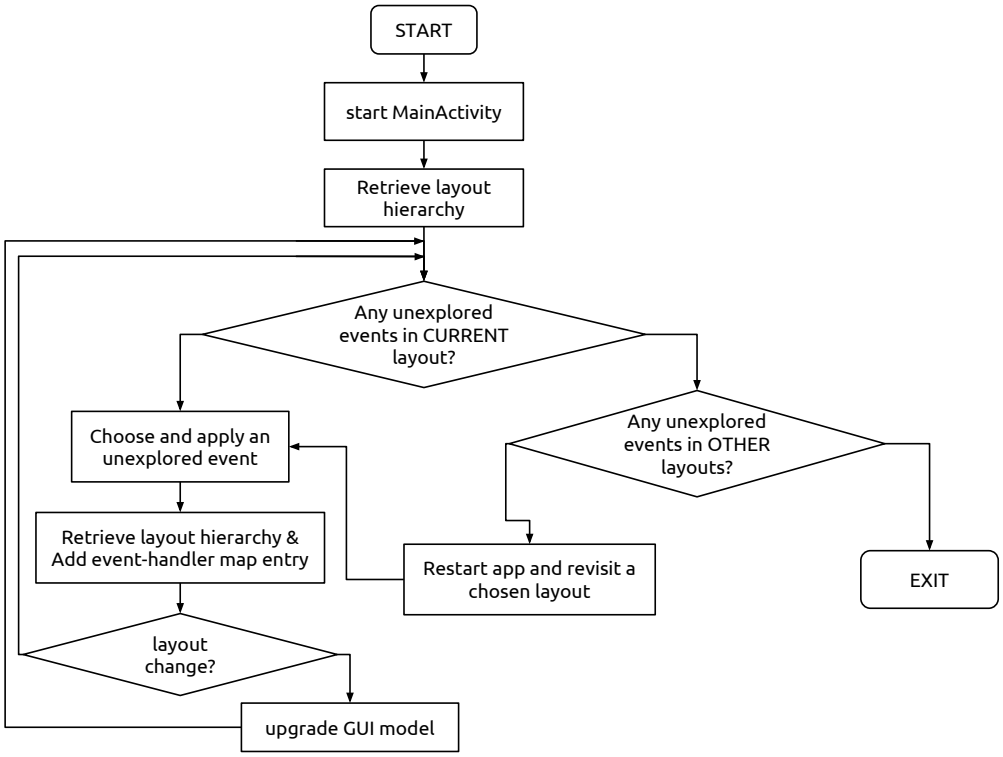


Figure 4: GUI Traversal Workflow

branch statements such as *if* and *switch*. The GUI traversal process has already concretely executed one path for each event handler method. For event handler methods that have multiple execution paths, the rest of the execution paths are executed symbolically. As a result, a *Path Summary* is generated for each execution path.

Path Summary contains 3 components: *execution log*, *symbolic states*, and *path constraints*. The *execution log* is the instruction sequence that were executed from the starting of an event handler to the returning of an event handler. The execution log includes not only the instructions in the event handler method, but also the instructions that were executed in nested method invocations. The *symbolic states* are the states of all the global variables at the end of the execution path. Since event handler methods in Android generally have one single parameter which is the *View* object correlating to the event, global variables (usually field members of global objects) are considered as *symbols*. The *path constraints* are a set of constraints that must be all satisfied in order to revisit a specific execution log. At the beginning of execution, the path constraints contain a single constraint *true*. When an *if* statement is executed during the execution, there are two different outcomes depending on whether the condition in the *if* statement is satisfied or not. When the direction of the *if* statement is decided, the corresponding constraint is added into the path constraints.

In our implementation, the *symbolic states* and *path constraints* are represented in the form of Abstract Syntax Trees (AST), using keywords to indicate symbols. The Dalvik bytecode instruction set contains 219 different instructions. Among them are many instructions that perform the same function but reflects different operand sizes or data types. For example, there are 7 instruc-

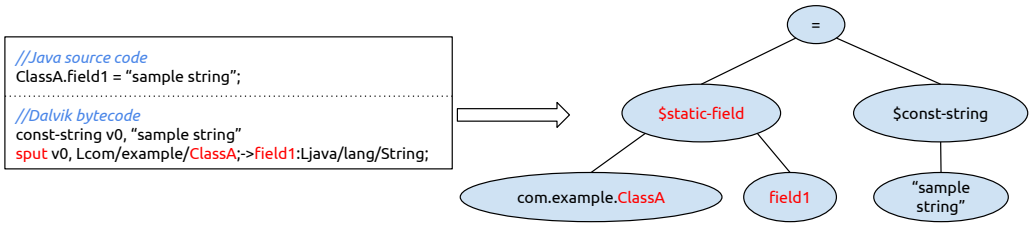


Figure 5: An Example of Symbolic State Expression Format

tions for loading an element from an array: *aget, aget-wide, aget-object, aget-boolean, aget-byte, aget-char, aget-short*. Our symbolic execution will parse these instructions using the same keyword *$aget*. Overall, we have created 17 different keywords for the whole Dalvik bytecode instruction set. Figure 5 shows an example of the symbolic state expression format of bytecode instruction *sput*, which writes the value of a static field. In this example, the root node of the AST has the name "=", indicating this expression is a symbolic state. The left child of the root node has a keyword *$static-field*, representing a symbolic value with a unique signature *com.example.ClassA.field1*.

The symbolic execution procedure takes a method signature as input, and outputs a list of path summaries for all the different execution paths. We have implemented the symbolic execution in a VM like structure. The symbolic VM contains heap and method stack. In the method stack, each method will be assigned a group of registers that stores local variables. The value stored in a register can be either a *literal value* or a *reference value*. Reference values usually represent the *address* of an object from the heap. We implemented the heap as a list of objects with a symbolic value and field members. At the end of each symbolic execution, the state of global variables are collected from the heap. Figure 6 shows an example of the symbolic VM state during runtime. In this example, *method1* is being executed and sits on top of the method stack. The instruction in line 0 writes literal value *0x5* into register v0. The instruction in line 1 creates an object in the heap, and puts the object's address *obj1* into register v1. The instructions in line 2 and 3 then copies the value of register v1 and write to two instance fields: *field1* and *field2* of the *this* instance of class *MainActivity*.

Unlike concolic execution in single entry point programs, concolic execution in Android programs is unable to "flip" a path constraint at the end of one execution and directly provide a concrete input for the next execution. The first reason is that Android programs have multiple entry points. The second reason is that the entry method parameters are not the only *symbolic values* that decide the path constraints. In order to find the symbolic values that can satisfy the "flipped" constraint, we often need to look into the path summaries of other event handlers and find the ones whose symbolic states at the end of execution can satisfy that constraint.

3.2.3 Event Sequence Generation

The event sequence generation takes specific code targets as input, and outputs a list of event sequences that can potentially reach the targets. First, a code target is matched with the execution logs of all the path summaries. If an execution log is found to contain the target, then the corresponding path summary can trigger this specific target. If the path summary is generated concretely, then

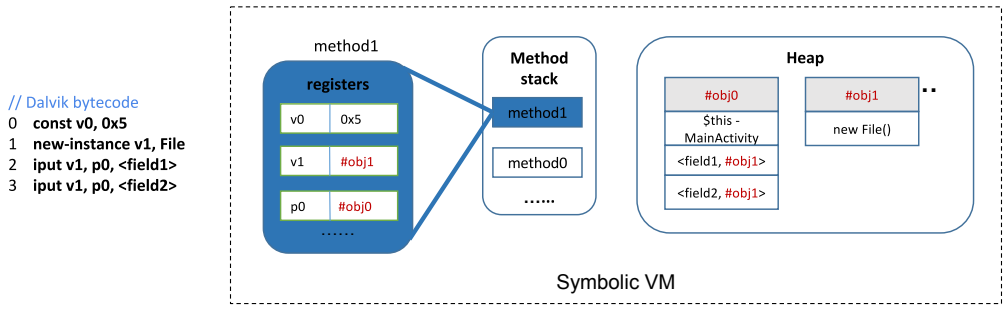


Figure 6: An Example of the APEX Symbolic Runtime VM State

the event sequence of this path summary is the input to execute the reflection call. But if the path summary is generated symbolically, a constraint solving process is required.

Constraint Solving starts from a desired path summary by adding preceding path summaries to the sequence, then keeps solving constraints of newly added path summaries until the main entry of the application is reached. Although a symbolically generated path summary does not have a concrete event sequence, the "final" event in its event sequence is already known. In order to find the preceding events, the path constraints of this path summary must all be satisfied.

Generally, the constraint solving process first searches for relevant symbolic states from other path summaries that can potentially satisfy each constraints, then the *CVC4* SMT solver is used to determine whether the relevant symbolic states satisfy the path constraints. When path summaries whose symbolic states satisfy all the path constraints are found, the event sequences of these path summaries is inserted in the front of the existing event sequence. These newly found path summaries then become the new subject of constraint solving. This process repeats until there are no new path constraints to solve. As a result, a list of event sequences are generated.

In practice, there are many types of path constraints that SMT solvers cannot directly solve. For instance, when system API *GregorianCalendar.get(Calendar.HoursOfDay)* is used in a branch statement, there are no GUI events that can satisfy this constraint. More generally, the two most common type of Android system APIs that we would encounter in path constraints are: (1) APIs for accessing OS settings and environment variables, (2) APIs for accessing GUI widget properties. For these types of APIs, we have developed a signature based solver that can generate a system event for recognizable API signatures in the constraint. This approach is implemented by manually building the signature pool and corresponding system events case by case.

Target Prioritization is used to deal with path explosion, a major challenge of symbolic execution. The number of execution paths in a method grows exponentially by the number of branching statements. In order to avoid path explosion, we implemented a target prioritization mechanism in the sequence generation process.

Each symbolic path summary is assigned with a priority factor. A path summary has higher priority if it meets any of the below conditions:

- its execution log contains the target
- its execution log contains GUI transition statements

- its symbolic states has more symbol variables

We implemented the priority with a linear calculation with a weight ratio of 1. This feature guarantees that more relevant path summaries are always solved first.

Sequence Validation is used to cull out infeasible paths. It is needed because during the sequence generation process, there is no validation on the event sequences. Some of the generated event sequences might be infeasible or incorrect, for reasons such as: errors in GUI model, unsolved constraint involving system APIs, etc. Therefore it is necessary to have a validation process.

Every generated event sequence is applied to the app while running on an Android device. We determine the correctness of a event sequence by comparing the concrete execution log to the corresponding symbolic path summary's execution log. This is implemented with instrumenting the beginning and ending of each basic block of every method. We can retrieve the concrete execution log via *logcat* during validation. If the concrete execution logs matches the corresponding symbolic path summary's execution log, then the event sequence is a correct sequence.

4.0 RESULTS AND DISCUSSIONS

In this section, we report the performance evaluation of JITANA and APEX.

4.1 Performance Evaluation of JITANA

To evaluate the performance of JITANA we applied it to five real-world apps. Table 3 lists these apps, together with data on their size and complexity.

On each of these apps, we measured the execution times required to perform (i) APK loading, (ii) call-graph generation, and (iii) reaching definitions analysis and addition of def-use edges to instruction graphs. We collected measurements for these tasks five times for each app; our results present averages across these five. We recorded execution times in seconds, and these included the times needed to process all system classes necessary to load the app classes.

Note that Facebook, unlike the other four apps, dynamically generates and loads secondary DEX files. JITANA is able to capture and analyze these files automatically using information from the virtual machine. Thus, the times reported for Facebook include the times required to analyze these secondary files.

Table 3 reports the three classes of execution times in the columns headed *"Loading"*, *"Call Graph"*, and *"R. Defs"*, respectively, along with the total time taken to perform all three (*Total*). These numbers were all gathered on a workstation with 3.2 GHz Core i5 and 32 GB DDR3 RAM running Apple OS X 10.11.3 (El Capitan).

As Table 3 shows, JITANA generated the basic analysis building blocks for the five apps in overall times ranging from 0.33 seconds for SuperDepth to 11.4 seconds for *Facebook*.

We also attempted to perform the same tasks using SOOT, but found that developing a methodology to perform a direct comparison was challenging. As such, we do not report numbers for SOOT in Table 3; instead, we report our observations and highlight the key differences that make a direct comparison difficult. First, JITANA was able to analyze more classes in four out of five apps. DEXPLER translates only classes that are part of the APK, and does not consider any system classes needed to initialize the apps. As such, we found that JITANA analyzed 4,942 classes for *Instagram* but DEXPLER passed only 4,641 classes on to SOOT to analyze. There were also differences in the numbers of analyzed classes in *SuperDepth*, *Google Earth*, and *Twitter*. These differences render

fair comparisons difficult. Second, DEXPLER does not consider dynamically generated classes, and *Facebook* generates three additional large DEX files as it initializes. In this case DEXPLER considers only the main DEX file which has 6,350 classes, whereas JITANA included all four DEX files, analyzing 23,621 classes. Again, this renders fair comparisons difficult.

Despite these difficulties, we do note the following. SOOT requires a translation process from DEX to *Jimple*, and this process alone requires more time than the entire analysis time required by JITANA. For example, it took DEXPLER 23 seconds just to translate the *Facebook* app and SOOT 21 more seconds to analyze the app with three missing dynamically generated DEX files. On the other hand, JITANA needed only 11.4 seconds to analyze all four DEX files in the app (and with almost four times as many classes to analyze).

Table 3. Analysis Times Measured When Applying JITANA to Five Real-World Apps

	# of				Time (seconds)			
Name	Classes	Methods	Fields	LoDC	Loading	Call Graph	R. Defs	Total
SuperDepth	68	2,035	1,355	50,779	0.16	0.05	0.12	0.33
Google Earth	1,213	10,698	4,137	136,679	0.58	0.15	0.28	1.01
Twitter	3,675	34,390	15,715	442,243	2.03	0.49	0.88	3.40
Instagram	4,942	37,747	16,514	477,700	2.11	0.53	0.94	3.58
Facebook (katana)	23,621	130,428	76,443	1,548,801	6.81	1.75	2.84	11.40

In summary, our investigation reveals differences between the two frameworks that can be summarized as follows. First, translation overhead can be high when SOOT is used to analyze Android apps. Second, the hybrid design of JITANA allows it to analyze more classes that include classes in the APK, system classes, and dynamically generated classes.

4.2 Performance Evaluation of APEX

We tested APEX on a total of 14 apps to examine the performance of APEX. The test apps include 12 apps from APAC engagements and 2 benchmark apps: *TippyTipper* and *Dragon*. These apps are mostly malware samples that utilize a variety of anti-analysis techniques to evade security analysis. We evaluated the effectiveness of APEX in terms of code coverage and target coverage using these subjects.

4.2.1 Code Coverage

Our code coverage is based on bytecode statement coverage. Comparing to method coverage, this fine grained metrics can better represent the percentage of different program paths being explored by our concolic execution engine. The total number of bytecode statements is measured statically using *apktool*. We instrumented these 11 apps and monitored *logcat* output during runtime to measure the number of covered bytecode instructions. Since most of the test apps contain third party libraries in their binaries, we have manually identified and excluded these library code from the results.

Table 4 shows APEX's code coverage results on the selected apps. The columns showed the total number of bytecode lines and the covered bytecode lines during the input generation. The last column showed the number of restarts during the GUI exploration stage. After manual examination

on the low coverage apps, the main reason for the low coverage is due to our constraint solver being unable to solve certain constraints involved with system APIs, for example, *Math.floor(double)* is not supported by the SMT solver. Effectively dealing with the system APIs and libraries is still a great challenge and continues to be the focus of APEX development.

Table 4. Code Coverage of APEX On 11 Apps

App Name	LoDC	Line Coverage	Restart Times
Dragon	335	307 (92%)	28
MunchLife	631	396 (53%)	8
TippyTipper	4520	2640 (58%)	20
CalcA	789	611 (77%)	2
CalcC	796	602 (76%)	2
CalcF	1210	663 (55%)	2
FullControl	3044	1229 (40%)	2
KitteyKittey	723	397 (55%)	3
PasswordSaver	842	478 (61%)	2
SMSBackup	778	478 (61%)	2
SourceViewer	382	245 (64%)	2

Generating Input for Specific Targets. To test APEX's effectiveness in generating input for specific targets, we picked 8 test apps and specified various code call sites within those apps as targets for APEX. The targets are determined using JITANA's coverage report from random testing and unit testing on the test apps. First we identified basic blocks that have not been executed; these methods represent hard to reach targets. We then selected the first instruction from each of those blocks and use those bytecode instructions as targets for APEX to generate input sequences.

Table 5. Target Coverage of APEX on 8 Apps

App Name	Targets Reached	Max Sequence Length
Dragon	5/5 (100%)	6
Munchlife	20/29 (69%)	8
TippyTipper	16/57 (28%)	5
BattleStat	10/88 (11%)	7
rLurker	12/141 (9%)	5
AudioSidekick	12/79 (15%)	4
AWeather	4/170 (2%)	3
Engologist	6/129 (4%)	3

If APEX is able to generate input sequences for a target and at lease one of the sequences are validated, then this target is considered to be reached. The results are shown in Table 5. The columns first showed the percentage of reached targets, then showed the number of events in the longest event sequences generated for each app.

The reason for the poor performance of APEX on some of the test apps is still mainly the unsolved API constraints. With our current signature based API constraint solver, we can only

deal with a limited set of APIs. Therefore, we could not complete the sequence generation process for the path summaries that contain APIs that we do not recognize.

Table 6. Comparison in Target Coverage Between *Collider* and APEx.

App Name	Target coverage by Collider	Target coverage by APEx
Tippytipper	7/16 (44%)	16/57 (28%)
Munchlife	6/10 (60%)	20/29 (69%)

Next we compare the target sequence generation result of APEX with that of *Collider*, a state-of-the-art concolic execution engine [12]. In the evaluation of *Collider*, the target lines were selected from unreached bytecode lines after running both *Monkey* and *crawler*. The targets used by *Collider* were not exactly the same as those used by APEX; however all targets were deemed hard to reach. In *TippyTipper*, *Collider* was able to reach 7 targets out of 16, while APEX has reached 16 out of 57. In *Munchlife*, *Collider* was able to reach 6 out of 10 targets, while APEX reached 20 out of 29. The comparison in target coverage is shown in Table 6. We can see that APEX has overall lower coverage rate while reaching more targets than *Collider*. Without a thorough comparisons, it is difficult to determine which tool performs better. However, *Collider*'s sequence generation requires a manually built GUI model, while APEX does not make any assumptions nor require manual effort with building the GUI model. Overall, despite the problems and limitations, APEX is easier to deploy than a state of the art concolic execution engine *Collider*, while able to reach more targets in the same apps.

5.0 APPLICATIONS OF PROPOSED FRAMEWORKS

In this section, we report the results of applying the proposed framework to address emerging security challenges. We investigated five scenarios in this study. In the first scenario, we explore an approach that uses runtime information to guide input generation through concolic execution. In the second scenario, we compare the performance of JITANA on an analysis task to that of a benchmark approach. We then present an example in which JITANA supports the creation of a real-time visualization engine to provide real-time feedback about the results of an analysis. In the fourth scenario, we investigate the scalability of JITANA and its potential applicability to bring-your-own-device (BYOD) environments. Finally, we illustrate how JITANA can perform analysis of dynamically loaded code.

5.1 Runtime Guided Input Generation through Concolic Execution

One of the main purposes of developing input generation techniques is to help dynamic malware analysis by generating input sequences to expose suspicious activities. In this application, we first try to validate a hypothesis that malicious code exists in rarely executed paths. We used JITANA to collect VM internal runtime information and identify rarely executed paths, then use these rarely executed paths in applications that APEX can support as the targets for the input generation process.

In order to identify rarely executed paths, we ran test apps on a Nexus 7 device with a modified Dalvik VM that are connected to workstations running JITANA. We used both *monkey* and unit

Table 7. Executed Malware Locations Across Engagements

Engagement	# of apps	# of known malware	# of executed malware	%
1A - 1C	18	16	9	56.0
3A - 3B	3	15	6	40.0
4A - 6A	3	6	3	50.0
Total	24	37	18	49.0

test cases to drive the execution of test apps. We configured the *monkey* to generate 20,000 random events from the main activity of each test app we then supplemented these random test cases with unit test cases to reach code coverage of 60% or more.

During the test, JITANA continuously receives the VM internal runtime information from the modified Dalvik VM, including the bytecode execution log and the bytecode traffic into the JIT compiler. JITANA overlays those dynamic information on top of static information, and generated a bytecode coverage report for the test app after each run. The coverage report is consisted of bytecode traces. Each trace contains a bytecode instruction's location (class signature, method signature, bytecode index) and an execution counter showing how many times this instruction has been executed.

With the JITANA coverage report, we first extract the bytecode traces with execution counter being 5 or fewer. These bytecodes are considered to belong in rarely executed paths. Furthermore, with the help from manual analysis and red reports of engagement apps, we highlight the known malicious code locations from the rarely executed bytecodes.

We first performed analysis across engagements. We report the result in Table 7. Even with our best efforts to provide good code coverage through random and unit testing, we are only able to executed 49% of malware locations in total. We also find that it is easier to generate input to get good code coverage for apps in Engagements 1A - 1C. The average code coverage for the 18 apps from the first engagement is 70%. On the other hand, it is much more challenging to generate inputs that can yield high code coverage for the remaining apps. In this case, in spite of our best efforts, we can only reach an average of 43%. This may indicate that these later apps are more complex.

Next, we selected applications that APEX can run without encountering any runtime errors. These apps are listed in 8 and are the same apps used to evaluate the effectiveness of APEX in Section 4.2. We report the number of malicious locations we retrieved from the red reports, then the total number of the malicious bytecodes that were executed. We can see that the malicious call sites in *FullControl*, *MorseCode*, and *smsBackup* are executed many times during both test runs. In these cases, *monkey* has executed the malicious call sites much more than unit test, due to the fact that *monkey* applied a much greater amount of events. On the other hand, unit test was able to trigger malicious activities while *monkey* could not for *CalcC*, *KitteyKittey*, *PasswordSaver*, and *SourceViewer*. The hardest to reach malicious call sites belong in *CalcA* and *CalcF* where neither *monkey* and unit test can trigger the malicious activities.

The rarely executed malicious call sites are then used as input for APEX to use concolic execution to generate input sequences. As explained in Section 3.2, during the input generation process, symbolic path summaries whose execution logs contain those bytecode instructions are high priority in the sequence generation process.

Table 8. Execution Counter of Malicious Call Sites

App name	# of malicious call sites	# of malicious call site execution (monkey)	# of malicious call site execution (unit test)
CalcA	1	0	0
CalcC	1	0	15
CalcF	1	0	0
FullControl	1	45	5
KitteyKittey	1	0	5
MorseCode	1	18	1
PasswordSaver	1	0	2
smsBackup	1	168	2
SourceViewer	1	0	4

We show the results of three apps: *CalcA*, *CalcF*, *SourceViewer* from the set in Table 8 as test apps. The malicious bytecode location are feed into APEX to see whether *APEx* can generate correct event sequences. The results are shown in Table 9. As we can see, APEX was able to reached the malicious call site in with an event sequence of length 6. Unfortunately, APEX did not reach the targets in the other two apps. The failure was due to the fact that In *CalcA* and *CalcF*, the malicious activities both contain API *GeorgianCalendar.get(hourOfDay)* which prevented APEX from generating correct event sequences. APEX was not able to generate sequences for targets in other apps as constraints to generate such sequences cannot be resolved.

Table 9. APEX Result in Malware Target Prioritized Input Generation

App name	Target reached by APEX	Max sequence length
CalcA	No	N/A
CalcF	No	N/A
SourceViewer	Yes	6

5.2 Inter-App Communication Analysis

Reusable components are an integral part of Android app development. There are four types of components in Android. (1) Activities are the user interface component of an app. (2) Services are used to run tasks that do not require any UI or tasks that are too long to run on the UI thread in the background. (3) Broadcast receivers are the components that can receive a message from any app. (4) Content providers work like databases and are used for sharing data between apps.

All but content provider components use *intents* to achieve inter-component communication. There are two types of intents: explicit and implicit. Explicit intents are designed specifically to cause a particular component to begin executing using its fully-qualified class name. Implicit intents specify actions but do not provide information on which component needs to run. An An-

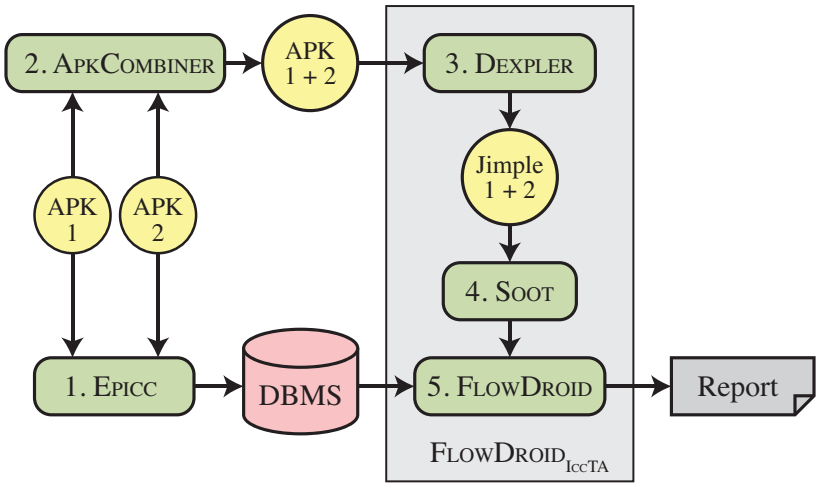


Figure 7: Workflow of IccTA Analyzing Two Apps

droidManifest.xml file defines intent-filters that help connect actions with components.[‖] Through these two types of intents, *Inter-Component Communication* (*ICC*) and *Inter-App Communication* (*IAC*) can occur. With IAC, data can flow from one app to another app. Being able to identify these communication channels can help engineers and analysts identify API incompatibilities after system or software updates, and vulnerable communication channels that cyber-criminals can exploit to compromise systems [17].

To detect information flow through IAC and ICC channels, Li et al. [17, 19] introduce IccTA, a Soot [15] based framework used to perform cross-app and cross-component taint analysis. Figure 7 highlights the workflow IccTA uses to perform IAC analysis, a workflow that includes several tools: Epicc [24], ApkCombiner [18], Dexpler [4] and FlowDroid [3]. First, Epicc stores information on edges that can potentially represent IAC connections in a database. Second, ApkCombiner combines multiple *Android Packages* (*APKs*) into a single DEX file. Next, to facilitate analysis by Soot, Dexpler converts DEX instructions to *Jimple* instructions. The combined apps are now analyzed by Soot to extract IAC edges and other data. Finally, FlowDroid builds complete control-flow graphs for the combined components using results from Soot and information stored in the DBMS by Epicc. It then performs taint analysis [19].

The foregoing process requires five tools to perform a workflow involving six steps (combining components, storing information, conversion, analysis, building control-flow graphs, and taint analysis), that require temporary data to be created by each step.

Next, we compared the performance of IccTA with that of Jitana for IAC analysis. Currently, however, Jitana does not yet support taint analysis itself so at best we can compare only the steps of the overall analysis that support the taint analysis step. However, because the processes of identifying edges and performing taint analysis are both done, in IccTA, using FlowDroid, we cannot remove just the taint analysis step. Therefore, we chose to remove the overhead of FlowDroid altogether. Since Jitana does perform control-flow analysis to detect IAC connections, our comparison errs in favor of IccTA.

Both Jitana and FlowDroid detect IACs due to implicit and explicit intents so we study apps

[‖] `http://developer.android.com/guide/components/intents-filters.html`

Table 10. Comparing Analysis Times and the Number of Discovered IAC Connections Between ICCTA and JITANA (Note That (–) Indicates That the Analysis Process Failed to Complete)

Applications	Source	ICCTA			JITANA		
		Time (seconds)	Implicit IACs	Explicit IACs	Time (seconds)	Implicit IACs	Explicit IACs
Echoer.apk StartActivityForResult1.apk SendSMS.apk (Total size = 760 KB)	DroidBench	88.2	2	0	8.6	2	0
Dragon.apk Morsecode.apk (Total size = 444 KB)	JitanaBench	53.0	0	1	7.6	0	1
App1_Source.apk App2_Sink.apk (Total size = 524 KB)	JitanaBench	(–)	(–)	(–)	9.0	1	0
com.facebook.katana-2.apk com.facebook.orca-1.apk com.spotify.music.apk (Total size = 80 MB)	Play Store	(–)	(–)	(–)	35.4	8	0
6 soc. network apps (Total size = 150 MB)	Play Store	(–)	(–)	(–)	82.4	39	0
13 games (Total size = 585 MB)	Play Store	(–)	(–)	(–)	191.4	0	0
7 random apps (Total size = 127 MB)	Play Store	(–)	(–)	(–)	120.2	4	0
Combine all 26 apps (Total size = 860 MB)	Play Store	(–)	(–)	(–)	(–)	(–)	(–)

that produce both types of IAC connections. We also added 26 apps randomly selected from the list of Google Play store's top-100 apps. These additional apps include social-networking, game, and other apps. Table 10 provides details on the apps, organizing them into eight groups (each represented by a row in the table). We apply each of the approaches to each group of apps to collect performance data when all apps within a group are analyzed simultaneously. This means that for ICCTA we attempted to use APKCOMBINER to combine all apps within a group, and for JITANA we attempted to use the CLVM to load the apps within a group.

Columns 3–5 of Table 10 show the results obtained using ICCTA, and Columns 6–8 show results obtained using JITANA. We present results using three metrics: the time required in seconds to perform the analysis and the numbers of implicit and explicit IAC connections detected. Entries of the form "(-)" indicate cases in which the approach was unable to perform the given analysis. As the table shows, ICCTA was able to analyze only the first two groups consisting of microbenchmarks. It also failed to detect IAC connections in Group 3 (Row 3), which consists of JITANA microbenchmarks. On the microbenchmark apps on which it functioned fully, it took 88.2 seconds and 53.0 seconds to analyze the programs. We also find that existing problems in ICCTA's components become limitations. For example, APKCOMBINER has been evaluated only on components smaller than 1.4 MB in size [18], and required 200 to 400 seconds. When applied to larger apps such as *Facebook* and *Spotify*, it failed. As also noted by Li et al [18], APKCOMBINER does not guarantee correctness of the combined file. This also becomes a limitation for determining the number of components that can be combined and analyzed by ICCTA. It also failed to analyze groups that use

real-world apps from Play store due to their large sizes [18]. For cases in which both approaches work, JITANA was between 7 and 10 times faster than ICCTA, even though JITANA was computing complete control-flow graphs and using them to construct IAC graphs.

5.3 In Situ Visualization of Code Coverage

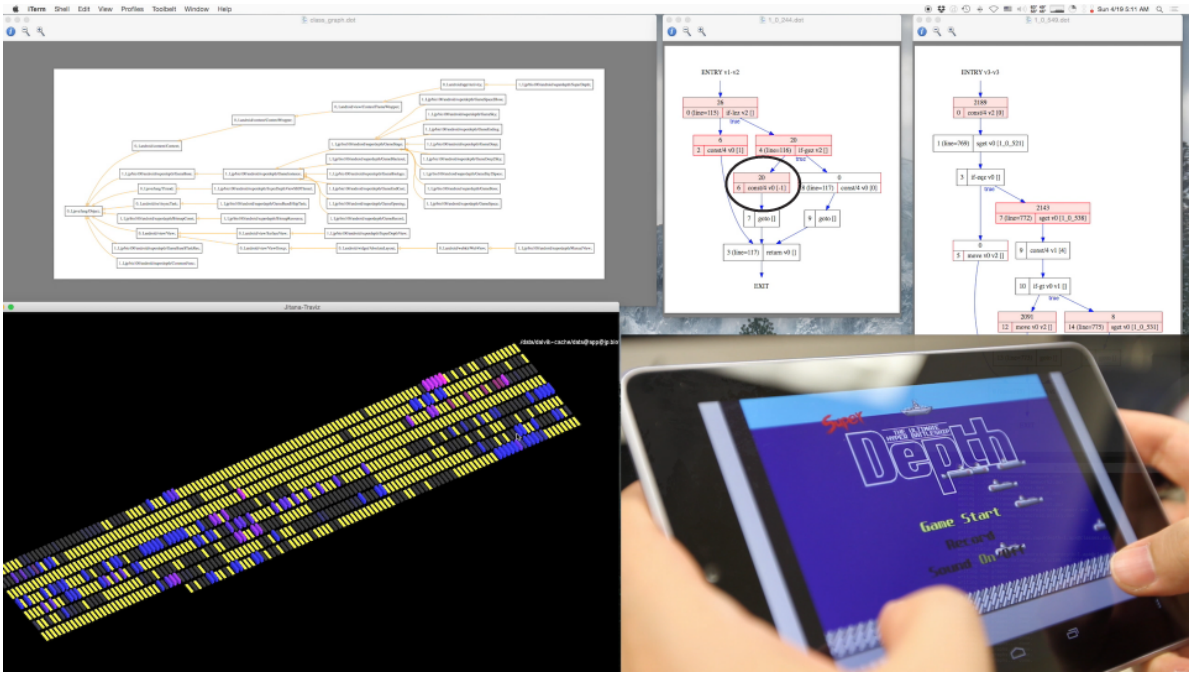


Figure 8: In-Situ Visualization with TRAVIS

Our third use case involves using JITANA to provide real-time feedback of code coverage measurement. Code coverage is an important metric for assessing the quality of test suites. Because code coverage is measured as a program is exercised, measuring it is a form of dynamic analysis. To measure code coverage in Android, EMMA** is still commonly used. EMMA was initially created as a code coverage tool for Java, but it now works for Android too. It supports both targeted unit testing and random testing using MONKEY, an Android UI exerciser.†† When used, it adds between 5% and 20% overhead to code execution time for Java programs. For event-based Android apps, our preliminary investigation using 10 apps reveals the overhead to be between 1% to 10% due to the necessary delays that must be added between two events. With this level of overhead, we do not expect JITANA to achieve significant performance benefits over EMMA on event-based apps. Instead, the main benefit of JITANA is in providing the ability for engineers and analysts to observe progress of the on-going analysis and monitor the intermediate results of code coverage measurement.

With JITANA, runtime information (basic block coverage in the case considered) is generated and processed immediately. Thus, there is an opportunity for attempting to visualize coverage information *in situ*. This section describes the process we followed to create a visualization tool,

**`http://emma.sourceforge.net/`.

††`http://developer.android.com/tools/help/monkey.html`.

TRAVIS, inside JITANA. We then illustrate the capabilities of TRAVIS and show how it can process the execution information sent by Dalvik to provide code coverage feedback.

Via a JDWP connection, TRAVIS periodically receives the dynamic execution information necessary to visualize traces from the Dalvik VM which was modified as described in Section 3.1.5. As soon as a DEX file on the device is loaded on the virtual machine, TRAVIS is notified with the file name. Upon notification, TRAVIS does the following: (i) copy the loaded DEX file from the device to the workstation using an `adb pull` command; (ii) create a buffer to store counter values for the DEX file; and (iii) let JITANA load the DEX file to update the VM graphs.

TRAVIS also polls for counter values every 50 milliseconds. The values are sent as an array of pairs of an instruction offset and the number of times that instruction has been executed since the last poll. The counter values are accumulated in the buffer created when the DEX file is loaded. The instruction graphs are updated with the new counter values from this buffer. This data is presented as traces on a screen with OpenGL renderings, and as instruction graphs rendered in a GRAPHVIZ viewer.

Figure 8 illustrates how TRAVIS can be used. A device (Nexus 7 in this case) is first connected to a workstation. Runtime information is sent from the device to a workstation running JITANA. In the figure, a person is playing SuperDepth, a classic video game (shown in the lower right quadrant). While the user plays, Dalvik sends execution information on-the-fly to JITANA, which processes the information to calculate code coverage, which is then fed to TRAVIS. The app requires no instrumentation.

In Figure 8, the upper left quadrant displays the method graph for *SuperDepth* and the upper right quadrant displays two instruction graphs. Shaded boxes indicate entry instructions in basic blocks. In each such box, there is also a counter to indicate the number of times that the basic block has been executed. For example, the block highlighted by an ellipse has been executed 20 times. The block above that corresponds to a conditional statement and so far, all decisions have taken the left branch. Note that these counters are continuously updated.

The bottom left quadrant shows the output of TRAVIS. Each small rectangle on what looks like a "keyboard" in the figure represents a basic block. On a color display (of this paper or of the output of TRAVIS), yellow rectangles indicate basic blocks that have not yet been executed, blue rectangles indicate "hot" basic blocks (i.e., basic blocks that have been executed more than five times), magenta rectangles indicate basic blocks that are currently being executed, and black rectangles indicate basic blocks that have been executed fewer than five times. (On a black-and-white printout, the colors range from dark gray to light gray with two intermediate shades.) The video clip that the images have been captured from is available at `https://www.youtube.com/watch?v=sPdrLdIKDx4`.

5.4 Device Analysis in BYOD Environments

To evaluate the scalability of JITANA, we also attempted to analyze all apps within devices simultaneously when analyzing apps for IAC connections. To do this, we selected three devices from our Android tablet stock pile. The first device contains 83 apps, the second contains 90 apps, and the third contains 106 apps. We pulled all the APKs from a given device into JITANA, using CLVM to load all the apps simultaneously and construct the graphs needed to build IAC graphs to detect connections. We anticipate that this particular task is an example of an analysis that could be useful for vetting devices in organizations that promote bring-your-own-device (BYOD) policies.

In such situations, instead of considering an app as the unit of analysis, we consider a device as the unit of analysis.

Table 11. IAC Analysis Results for Three Devices

Device ID	# of Apps	Total Size (MB)	Edges (Implicit)	Edges (Explicit)	time (seconds)
1	83	129	476	21	136
2	90	438	1216	50	601
3	106	1298	(–)	(–)	(–)

Table 11 reports the results of this investigation. As shown, JITANA was able to analyze two of the devices (devices with 83 and 90 apps) successfully. The time required to perform these two analyses was 136 seconds and 601 seconds, respectively. Note that each reported time includes the time needed to load the apps from the device to the workstation running JITANA. Our system ran out of memory for the device with 106 apps. Nevertheless, the results on the first two devices suggest that JITANA can support larger-scale analyses by using larger computer clusters. Further, by using BGL, JITANA should be able to perform analyses using any machines that provide BGL support.

5.5 Analysis of Reflection Usage in Android Apps

In Java programming language, which is the main programming language for Android application development, *reflection* provides a computer program with the ability to load certain classes during execution [27]. The main goal is to allow a program to examine and modify its structure and behavior dynamically. As such, reflection is one important mechanism that developers use to achieve *backward compatibility* [2]. In addition, *reflection* is also a powerful tool that can help with debugging as well as developing pluggable code.

However, the dynamic property of reflection has also been exploited by malware authors to obscure intentions or hide malicious payloads from malware analysis tools. Typical malware analysis tools that analyze source code or bytecode to detect vulnerabilities tend to have trouble analyzing reflection target classes due to multiple reasons. First, while static analysis can be commonly used to identify where reflection calls are being made, the dynamic nature of reflection requires that analysts have the input sequences that can exercise these reflection call sites. However, creating precise sequences that can reach specific targets in event-based and GUI rich applications is still quite challenging.

To better understand how reflection is used as part of Android app development, we investigate its usage in real-world Android applications. To do so, we collected nearly 1800 Android app samples. We divide the apps into 3 groups, as shown in Table 12 below:

The Android application samples consist of: 1258 malware samples from Android Malware Genome Project (AMGP) [31], 378 newer (after 2012) malware samples collected from various sources, and 126 popular Android apps in 2014 that have been downloaded from Google Play Store. To determine reflection usage, we first implemented a static analysis in Soot based on the idea introduced by Bodden et al. [5]. We also utilized Apktool [29], a reverse engineering tool for Android apps, to help accomplish this task. A *Reflection Information Table* is generated by reflection logger to record each identified reflection call site.

Table 12. Android Application Sample Groups

app sample group	total #	time
Android Malware Genome Project	1258	2010-2012
Newer Malware	378	after 2012
Google Play Store Top Chart	126	2014

We then classified reflection into four types as shown in Table 13 based on different Java semantics. Typically, method invocation via reflection involves the following APIs:

- Class.forName,
- Class.getDeclaredMethod (or Class.getMethod)
- Method.invoke.

The name parameter used in *forName()* or *getDeclaredMethod()* can either be a constant string or a string variable. The class object in a *Class.forName()* call is found by either the default class loader or a custom class loader.

Table 13. Reflection Classification

Category	Reflection Target	Class Loader
1(a)	Constant string	Default
1(b)	Constant string	Custom
2(a)	String variable	Default
2(b)	String variable	Custom

Each type of reflection call indicates different techniques required to identify targets. An example of Type 1(a) is shown in Figure 9. Method member *"MyMethod"* of class *"MyClass"* is invoked via reflection APIs. The reflection target, i.e., the string names, are constant strings. The default system ClassLoader is designated at runtime to look for Class *"MyClass"*. Determining targets for this type of reflection call only requires simple static analysis. The string names are already known, and the binaries of class *"MyClass"* can only come from the *classes.dex* within the APK file or system libraries.

```
Class cl = Class.forName("MyClass");
Object obj = cl.newInstance();
Method m = cl.getDeclaredMethod("MyMethod");
m.invoke(obj);
```

Figure 9: Reflection Type 1(a)

An example of Type 1(b) is shown in Figure 10. In this example, the reflection target is still constant strings, same as Type 1(a). However, additional parameters are used in the *Class.forName()* call. The third parameter *loader* is a custom class loader object. A custom class loader can specify an arbitrary path to load classes at runtime. As such, it is possible that the class binaries are placed

```
DexClassLoader loader = new DexClassLoader(
libpath,dir,null,getloader() );
Class cl =Class.forName("MyClass",true,loader);
Object obj = cl.newInstance();
Method m = cl.getDeclaredMethod("MyMethod");
m.invoke(obj);
```

Figure 10: Reflection Type 1(b)

outside of the system library and *classes.dex* from the APK file. This type of reflection call would require dynamic analysis approaches to precisely determine targets.

An example of Type 2(a) is shown in Figure 11. The target names in *Class.forName()* and *Class.getDeclaredMethod()* are provided as string variables, and default class loader is used in *Class.forName()* call. It is guaranteed that the class is loaded from the system library path by the system class loader. However, in order to retrieve the class name or the method name, dynamic analysis approaches are needed to precisely determine targets.

```
Class cl = Class.forName(className);
Object obj = cl.newInstance();
Method m = cl.getDeclaredMethod(methodName);
m.invoke(obj);
```

Figure 11: Reflection Type 2(a)

An example of Type 2(b) is shown in Figure 12. In this example, the reflection target name is provided as a string variable. A custom class loader is also used in *Class.forName()* to search and load the target class. As such, static analysis alone is not enough to find the class name or library path of the class loader, especially when runtime data such as user input is involved. This type of reflection calls require dynamic analysis approaches to determine target information.

```
DexClassLoader loader = new
DexClassLoader(libpath,Dir,null,getloader());
Class cl =Class.forName(className,true,loader);
Object obj = cl.newInstance();
Method m = cl.getDeclaredMethod(methodName);
m.invoke(obj);
```

Figure 12: Reflection Type 2(b)

We ran our reflection analysis on every application in those three groups, calculate the number of each reflection type. The results for the AMGP malware and newer malware are shown in Table 14. As we can see, 78.7% of reflection calls in AMGP malware belongs to type 1(a); static analysis should be able to effectively determine reflection targets. We also find that 20% of reflection calls belong to 2(a), which means that these reflection calls use string variables to specify invocation targets. This type of reflection calls require dynamic analysis approach to solve them.

On the other hand, in the newer malware group, there are 20% reflection calls belong to 1(a), and 76% belongs to 2(a). In both malware groups, there are not many cases of 1(b) and 2(b) where

Table 14. Reflection Usage in Malware Samples

Malware sample group	AMGP		Newer Malware	
Total number of APKs	1258		378	
Year	2010-2011		2012-2014	
Reflection Classification	Number of reflection calls	Percentage	Number of reflection calls	Percentage
1(a)	6357	78.7%	946	20%
1(b)	2	0.02%	7	0.1%
2(a)	1664	20.6%	3613	76.2%
2(b)	56	0.7%	176	3.7%

custom class loader is involved. However, we can observe the trend of increasingly percentage of type 2(a) reflection calls.

The results of Play Store Apps are shown In Table 15:

Table 15. Reflection Usage in Play Store Apps

Total number of APKs	126	
Year	2014	
Reflection Classification	Number of reflection calls	Percentage
1(a)	1774	53.9%
1(b)	31	0.9%
2(a)	1315	40%
2(b)	170	5.2%

In these 126 top downloaded Play Store Apps, we can see a similarity to the malware sample groups: Reflection types 1(a) and 2(a) have the highest percentages out of the four categories. However, we do notice that in Play Store Apps, more type 2(b) reflection calls are used than the malware sample groups.

Considering the fact that the number of reflection calls is also related to the number of samples within each group, we decide to calculate the reflection density of each sample group, in order to observe the trend of reflection usage in these samples. First, we calculate the reflection density per class, as shown in Table 16 below:

As shown, Play Store Apps have the highest number of classes and highest number of classes that contain reflection calls. However, the newer malware group has the highest reflection density per class.

The reflection density per method is shown in Table 17. Similar to the reflection density per class, the Play Store Apps have the highest number of methods and methods containing reflection calls. However the newer malware group again has the highest reflection density per method.

In summary, while the percentages of of reflection usage remain relatively flat among the three groups of applications (1.01% to 1.67%), we see that modern Android applications (newer malware

Table 16. Reflection Density per Class

Sample Collection	# of classes	# of classes with reflection calls	Density (per class)
AMGP	299,126	3,015	1.01%
newer malware	118,267	1,975	1.67%
Play Store Apps	372,985	4,317	1.16%

Table 17. Reflection Density per Method

Sample Collection	# of methods	# of methods with reflection calls	Density (per method)
AMGP	1,718,380	4,494	0.26%
newer malware	839,109	3,248	0.39%
Play Store Apps	2,364,347	7,423	0.31%

and Play Store apps) are increasingly using more of Type 2(a) reflection calls, in which targets cannot be determined through static analysis alone. By being able to capture runtime information, JITANA can be used to identify these reflection targets. APEX can also be used to generate event sequences that can reach these reflection calls.

Recently, we have also seen more usage of Type 2(b) in modern apps. Furthermore, a recent technique for removing reflection code after each run (e.g., a commonly used mechanism to serve advertisements [13]) can also be used to deliver malicious code and then delete it after it has been executed. To prevent later retrieval by analysts, attackers can also move the code from an original downloading site after it has been downloaded. By working closely with Dalvik, we are able to extend the mechanism used in TRAVIS to cache dynamically loaded classes for analysis. This feature is critical for security analysts who need to detect malicious payload that may be hiding as reflective code and software updates.

6.0 FUTURE WORK

We have been developing JITANA and APEX over the past two years. We plan to officially release the source code and binaries of both frameworks under a BSD license in July, 2016. At that time, we will also include additional tools that we are currently developing. Next, we discuss on-going efforts to produce additional tools.

As shown in our BYOD use-case, when a large number of apps are used, JITANA can experience out-of-memory errors when run on a desktop or laptop. Because our approach is based on BGL, we are developing approaches for partitioning the processes used to generate graphs and perform analysis so that they can be performed in parallel on high-performance computing clusters. On the other hand, because JITANA incurs low overhead when used to analyze small numbers of apps, we plan to create a version that can run directly on an Android device to perform real-time analysis as an app is downloaded and then perform light-weight analysis and monitoring as apps run.

Currently, techniques such as TAMIFLEX and HARVESTER [5, 25] can detect reflective code by

instrumenting apps to report reflection targets and then capturing them off-line. However, a recent technique for removing reflection code after each run (e.g., a commonly used mechanism to serve advertisements [13]) can also be used to deliver malicious code and then delete it after it has been executed. To prevent later retrieval by analysts, attackers can also move the code from an original downloading site after it has been downloaded. By working closely with Dalvik, we are able to extend the mechanism used in TRAVIS to cache dynamically loaded classes for analysis. This feature is critical for security analysts who need to detect malicious payload that may be hiding as reflective code and software updates.

We have began an effort to integrate APEX with JITANA. We currently have a basic implementation of a symbolic execution engine in JITANA that can work on clusters and we are refining the implementation to take advantage of existing constraint optimization frameworks to reduce runtime overhead. We are also developing a taint analysis engine for JITANA. Finally, as Google has already shifted from Dalvik VM to *Android Run Time* (*ART*), one of our priorities is to make JITANA work with ART. We have already analyzed the structure of ART and have determined how to capture runtime information that can be used by JITANA to perform dynamic analysis. We are also working to extend JITANA to support analysis of binary components that can interface with Android applications through Java Native Interfaces (JNIs).

7.0 CONCLUSION

We have made three contributions that advance the state-of-the-art in program analysis. First, by harnessing the power of generic programming and exploiting runtime events and information, we have built a highly efficient hybrid program analysis framework that is capable of expanding the analysis scope to cover most apps installed on an Android device. We provide common analysis building blocks that include control-flow, data-flow, and points-to analysis engines. Our evaluation results and use cases show that JITANA can analyze more classes (e.g., system related and dynamically generated classes) in much less time than SOOT. We also discussed on-going work to further extend the capabilities of JITANA that includes supporting parallel analysis on clusters, migration to ART, and caching and analyzing dynamically loaded code.

Second, we developed APEX, a concolic execution based event sequence generator that produces complete GUI models and identifies paths and connections to the GUI models that can be used to generate event sequences to reach specific targets in a program. We applied JITANA to validate a hypothesis that malicious code exists in rarely executed paths. We found that in many engagement apps, half of the malicious locations are hard to reach using random and unit testing. We then used APEX to generate event sequences to reach those targets. Unfortunately, these hard to reach targets often involve calls to libraries and system APIs that are not fully supported by our concolic execution engines. As such, we were not able to generate sequences to reach targets in many apps.

Third, we showed through five examples how the proposed frameworks can be used to address emerging security challenges. We have shown that by using the proposed frameworks, event sequences can be generated to exercise hard-to-reach targets. Complex analyses such as IAC detection can be quickly developed and effectively and efficiently performed to address emerging security needs such as vetting devices in BYOD environments and detecting malicious apps that collude. It can also analyze a large number of apps concurrently (it has successfully analyzed as many as 90 apps concurrently) and can provide real-time feedback to engineers and analysts so

that they can evaluate the progress and effectiveness of an on-going analysis. We have also shown that it can efficiently handle dynamism of modern programming language including identify and analyze reflective methods.

8.0 ACRONYMS AND GLOSSARY

ART	Android Run-Time
APEx	Android Path Explorer
BYOD	Bring Your Own Device
CLVM	Class Loader Virtual Machine
IccTA	Inter-component communication Taint Analysis
Jitana	Just-In-Time Analysis
JNI	Java Native Interface
JVM	Java Virtual Machine
VM	Virtual Machine

REFERENCE

[1] S. Acharya. Google Removes 13 Android Apps from Play Store Infected with Brain Test Malware. http://www.ibtimes.co.uk/google-removes-13-android-apps-play-store-infected-brain-test-malware-1537049, January 2016.

[2] Android Developper Blog. Backward compatibility for android applications. `http://android-developers.blogspot.com/2009/04/backward-compatibility-for-android.html`, April 2009.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 2014.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, pages 27–38, 2012.

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the International Conference on Software Engineering*, pages 241–250, May 2011.

[6] R. Chirgwin. Uk universities, mcafee collude to beat collusion attacks. http://www.theregister.co.uk/2014/02/27/uk_unis_mcafee_collude_to_beat_collusion_attacks/, February 2014.

[7] David Bisson. Whoa! nearly 5,000 new android malware samples discovered each day in q1 2015. `http://https://grahamcluley.com/2015/07/nearly-5000-android-malware/`, July 2015.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. Conf. Prog. Lang. Des. Impl.*, pages 213–223, June 2005.

[9] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 291–310, October 2006.

[10] V.-V. Helppi. What Every App Developer Should Know About Android. http://www.smashingmagazine.com/2014/10/02/what-every-app-developer-should-know-about-android/, October 2014.

[11] Apple breaks new iphones with terrible software update. http://www.slate.com/blogs/future_tense/2014/09/24/apple_ios_8_0_1_software_update_major_bugs_hit_iphone_6_6_plus.html, 2014.

[12] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, Lugano, Switzerland, 2013.

[13] John Emulators. John-NES Lite. https://play.google.com/store/apps/details?id=com.johnemulators.johnneslite&hl=en, June 2015.

[14] Kaspersky. List of malicious android apps hits 10 million. `http://www.kaspersky.com/about/news/virus/2014/Number-of-the-week-list-of-malicious-Android/apps\-hits-10-million`, February 2014.

[15] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, April 2003.

[17] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th ACM/IEEE International Conference on Software Engineering*, pages 280–291, 2015.

[18] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. L. Traon. *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, chapter ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis, pages 513–527. Springer International Publishing, Cham, 2015.

[19] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis. *CoRR*, abs/1404.7431, 2014.

[20] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[21] K. Manikas and K. M. Hansen. Software ecosystems - a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[22] E. Messmer. Black Hat demo: Google Bouncer Can Be Beaten. http://www.networkworld.com/news/2012/072312-black-hat-google-bouncer-261048.html.

[23] J. Oberheide and C. Miller. Disecting The Android Bouncer. http://jon.oberheide.org/files/summercon12-bouncer.pdf.

[24] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *USENIX Security Symposium*, pages 543–558, 2013.

[25] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting Runtime Data in Android Applications for Identifying Malware and Enhancing Code Analysis. Technical Report TUD-CS-2015-0031, Technische Universitat Darmstadt, February 2015.

[26] M. J. Schwartz. Google Play Exploits Bypass Malware Checks. http://www.informationweek.com/security/application-security/google-play-exploits-bypass-malware-chec/240001691.

[27] D. Sosnoski. Java Programming Dynamic: Introducing Reflection. `http://www.ibm.com/developerworks/library/j-dyn0603/`, June 2003.

[28] A. Stepanov and P. McJones. *Elements of programming*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[29] R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files, 2012.

[30] YouTube API change: some older devices can't update to new app. http://hexus.net/ce/news/audio-visual/82570-youtube-api-change-older-devices-update-new-app/,
2014.

[31] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.